POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE

LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE

Distributed measurement and data acquisition systems

Prof. Ing. Attilio Di Nisio

Documentation title:

Position Control of Linear Actuator

Student:
Nicola Saltarelli

ANNO ACCADEMICO 2023-2024

Abstract

This documentation presents the implementation of a system for controlling the position of a linear actuator. The system comprises an HC-SR04 ultrasonic sensor, an ELEGOO UNO R3 processing unit, and a motor control board based on the Dual H-Bridge L298N driver.
The ultrasonic sensor plays a fundamental role in the system by providing feedback on the distance of the actuator from its reference position. The ELEGOO UNO R3 is responsible for acquiring and analyzing data from the sensor, generating PWM signals for controlling the actuator through the L298N driver.
Furthermore, to facilitate user interaction and control, an intuitive user interface has been developed using LabVIEW. This software provides operators with an efficient and user-friendly tool for accurately adjusting the actuator position, ensuring a smooth and reliable control experience.

Contents

1 Technical Background **3**
1.1 Linear actuators 3
1.2 Ultrasonic sensor 4

2 Experimental phase **5**
2.1 Circuit Connections Representation 5
2.2 Microcontroller Code 6
2.3 LabVIEW interface 7

3 Results **10**
3.1 Comparison between manual and automatic speed control 10
3.2 Performance analysis 11
3.3 Precision analysis 13

A Microcontroller code **15**

B Python Code **18**

1 Technical Background

In this section of the documentation, we will proceed with the exposition of the fundamental components, focusing on the linear actuator and the ultrasonic sensor. Each component will be addressed through a brief subsection aimed at providing a comprehensive overview of their role and usage within the project's context.

1.1 Linear actuators

Linear actuators represent a category of mechanical devices designed to convert rotary motion into linear motion, namely movement along a straight trajectory. This transformation can be achieved through the use of alternating current (AC) or direct current (DC) electric motors, or by employing hydraulic and pneumatic systems.
Among the various available options, electric linear actuators emerge as the preferred choice in numerous application contexts where precision and smoothness of motion are required. They find application in a wide range of equipment intended for operations such as tilting, lifting, pulling, or pushing with a specific applied force.
An electric actuator consists of three elements: a piston, a motor, and gears. The motor can be either alternating current (AC) or direct current (DC), depending on the desired power output or other factors.
Upon receiving a signal through a simple command, such as an on-off button, the motor converts electrical energy into mechanical energy by rotating the gears connected to the piston. This rotation allows the piston to move, enabling the sliding bushing to move either outward or inward depending on the signal received.

The actuator employed is a device manufactured by Justech, crafted from aluminum alloy and boasting an IP54 protection rating, providing significant resistance to dust and water ingress. This actuator is designated as a power unit, featuring a length of 100 mm (equivalent to 4 inches). Its displacement capability is swiftly executed at a speed of 4 mm per second, while requiring a 12V DC input voltage.
In terms of load capacity, it can withstand up to 1500 N (equivalent to 3300 pounds). Its nominal power is 20 W, though it can reach a maximum of 30 W. It is engineered to operate with low noise, registering less than 42 dB during operation.

Figure 1: Used linear actuator.

1.2 Ultrasonic sensor

Ultrasonic sensors operate on the principle of "time of flight," which involves measuring the time taken by a sound signal to reach an object and return to the sensor. In the specific case of the HC-SR04 sensor, the emitted ultrasonic pulse has a frequency of approximately 40 kHz, and the time of flight is measured in microseconds. As the operating voltage is 5V, the sensor can be powered directly via Arduino.

The HC-SR04 sensor features four pins: Vcc (+5V), Trigger, Echo, and GND. To activate the sensor, a high pulse of at least 10 microseconds must be sent to the Trigger pin. At this point, the sensor emits the sound signal and waits for the return of the reflected waves; the duration of this return is indicated on the Echo pin with a high pulse.

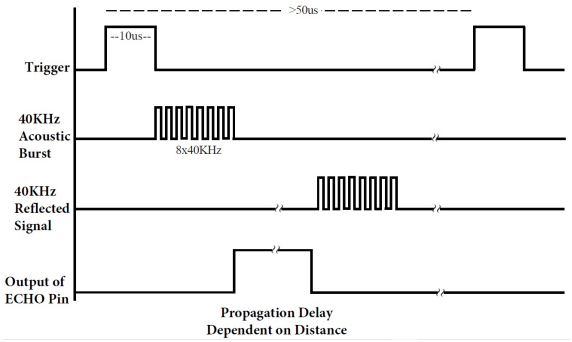

Figure 2: HC-SR04 operating principle.

If the duration of the Echo signal exceeds 38 ms, it is assumed that there are no obstacles in the path of the signal (time measurements greater than 38 ms are not considered reliable). To ensure the proper execution of subsequent measurements, it is advisable to wait for an interval of 50-60 ms before querying the sensor again to avoid interference.

To convert the measured time into distance, it is necessary to consider that the speed of sound varies depending on the ambient temperature. Assuming a temperature of 20°C, the speed of sound is approximately 343 m/s. Since sound travels twice the distance to be measured (out and back), the total distance traveled can be calculated taking this duplication into account.

Figure 3: Ultrasonic sensor HC-SR04.

2 Experimental phase

The experimental phase of the project is characterized by the development of code intended for implementation on the microcontroller to manage its operational logic, and the creation of a graphical interface designed to communicate with the microcontroller through the serial port. After completing the implementation of the code on the microcontroller, a user interface was subsequently developed to facilitate interaction between users and the microcontroller itself. This interface allows for control of the linear actuator position through an environment developed in LabVIEW, thus providing an intuitive and accessible means of managing the device's functionalities. Therefore, in this section, the implementation of the code on the microcontroller will be presented first, followed by an explanation of the user interface.

2.1 Circuit Connections Representation

This subsection presents the representation of the circuit connections required for the implementation of the linear actuator position control project. Figure 4 illustrates the connections between the HC-SR04 ultrasonic sensor, the processing unit ELEGOO UNO R3, the motor control board utilizing the Dual H-Bridge L298N driver, and the linear actuator used.

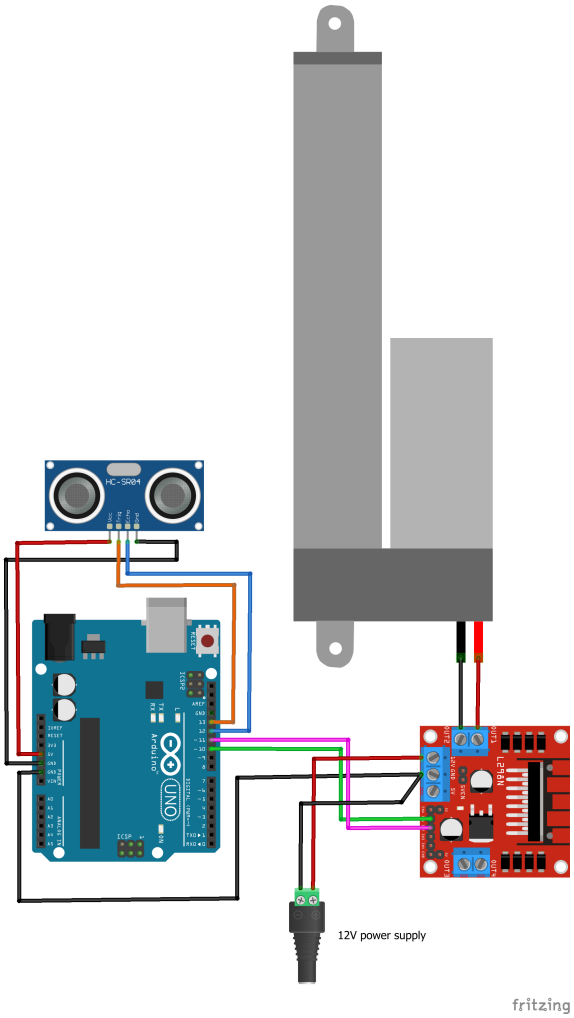

Figure 4: Circuit connections.

2.2 Microcontroller Code

The code implemented on the microcontroller follows a well-defined operational logic. Upon receiving the target position from the user, the system validates whether it falls within the prescribed limits. If the target position is valid, the system calculates the current distance between the actuator and the ultrasonic sensor. Subsequently, it writes to the serial port the calculated distance value and the time taken for the measurement. At this juncture, the actuator extends or retracts as per the target position, sustaining this movement until the measured distance exceeds the sum of the tolerance and the target position itself. It is noteworthy that the user can adjust the actuator's movement speed by sending speed values over the serial port. Upon the measured distance no longer exceeding the mentioned sum, the actuator ceases movement, awaiting a new target position input from the user. This process is schematically represented in the flowchart shown in Figure 5.

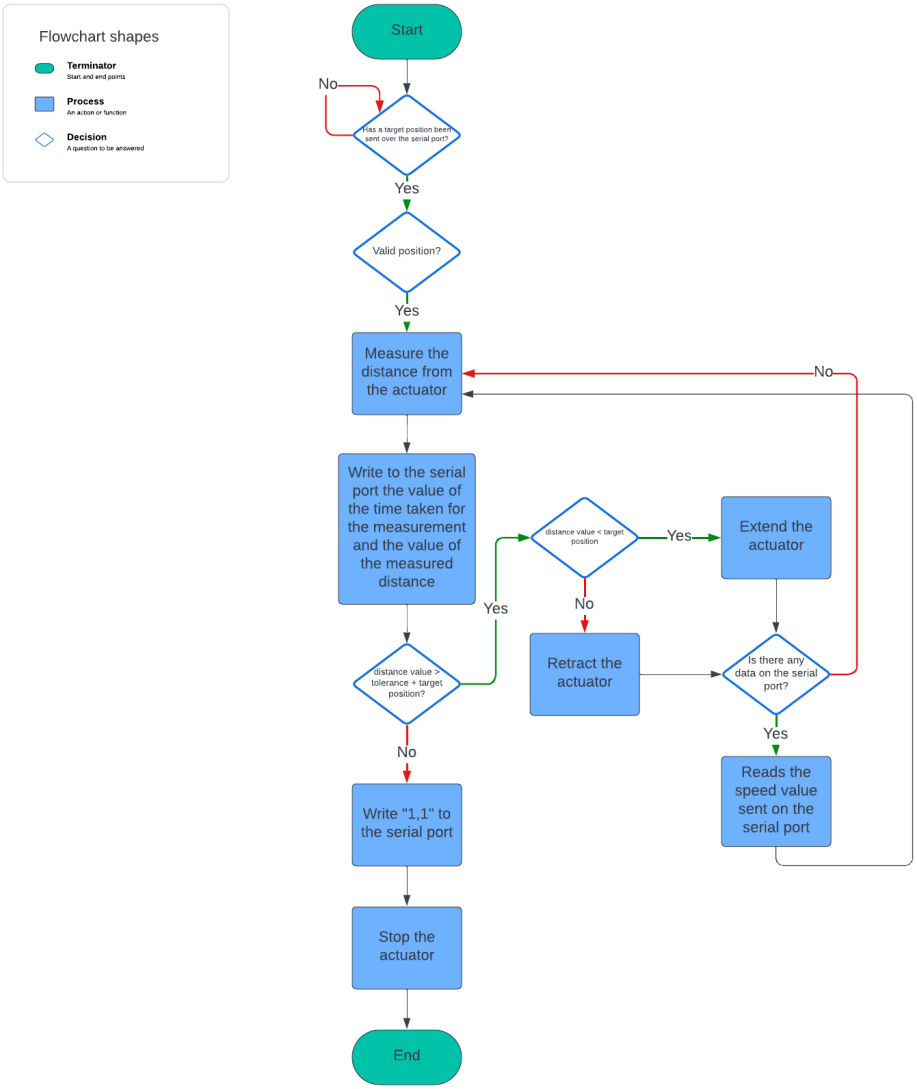

Figure 5: Microcontroller code flowchart.

2.3 LabVIEW interface

In order to provide a comprehensive and intuitive description of the user interface developed in LabVIEW, reference is made to Figure 6.

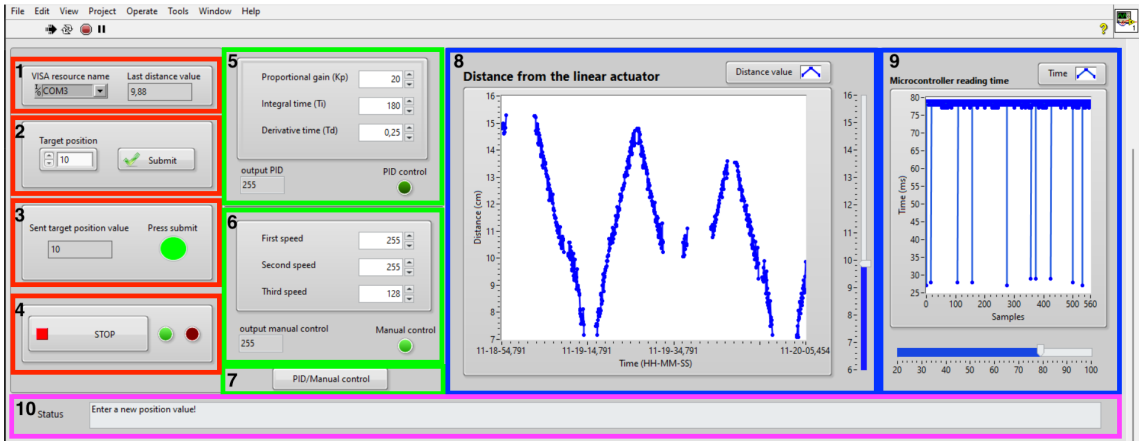

Figure 6: LabVIEW interface.

As illustrated in Figure 6, the interface has been divided into 10 components, each of which will now be described in detail:

- Component Number 1: This component hosts the "VISA resource name" block, which allows specifying the resource for opening a VISA session. Additionally, it includes a block that displays the last distance value read.
- Component Number 2: Used to transmit the target position provided by the user, this component includes a block for entering the target position and a submit button.
- Component Number 3: Comprising a indicator on the left displaying the value of the target position sent and to be reached. On the right, there is an LED that activates if the user can enter a new target position, remaining off if the user must wait for the program to execute.
- Component Number 4: Intended for interrupting the program execution, this component includes a stop button and two LEDs: one green, indicating program execution, and one red, signaling activation of the stop button. The green LED lights up as soon as communication with the microcontroller is possible, remaining on until the stop button is pressed.
- Component Number 5: Allows specifying PID controller parameters and displays the last calculated control action value. It also includes an LED that lights up when the automatic control mode is active.
- Component Number 6: Enables manual control of the actuator motor speed, offering three different speeds based on the reached position. It includes a block to display the last speed value sent to the microcontroller and includes an LED that lights up when the manual control mode is active.
- Component Number 7: Consists of a button used to enable one of the two functionalities of components 5 and 6.
- Component Numbers 8 and 9: Contains graphs representing, respectively, the position values calculated by the microcontroller and the time taken for the measurement.

- Component Number 10: Indicates the program status, providing messages to guide its execution.

The following description outlines the operational logic of the developed LabVIEW interface. Once the program is initiated by pressing the "Start" button, it establishes a connection to the serial port via NI-VISA, an API providing a programming interface for controlling serial instruments in NI application development environments such as LabVIEW.
Subsequently, the program runs cyclically until the user presses the stop button. During execution, a check is performed to determine if a new position value can be sent. If the "Press Submit" LED is on, it indicates that the program is awaiting user input of a new position value.
Once the user presses the "Submit" button, the selected value is transmitted over the serial port, and the "Press Submit" value is set to false, thereby preventing further writing of position values to the serial port.
Next, the program reads data sent by the microcontroller via the serial port. In case of warning signal sent by the microcontroller ("1,1" if the actuator has reached the desired position), the "Press Submit" value is set to true, allowing the user to input a new position value.
Otherwise, if a value other than "1,1" is read from the serial port, corresponding to position values and the time taken to read them from the microcontroller, these values are used to update the graphs. Subsequently, if the PID control mode is selected, the velocity value calculated by the PID.vi block in LabVIEW is written to the serial port. Otherwise, the velocity chosen by the user is written.
In Figure 7, the flowchart summarizing the described operational logic is depicted.

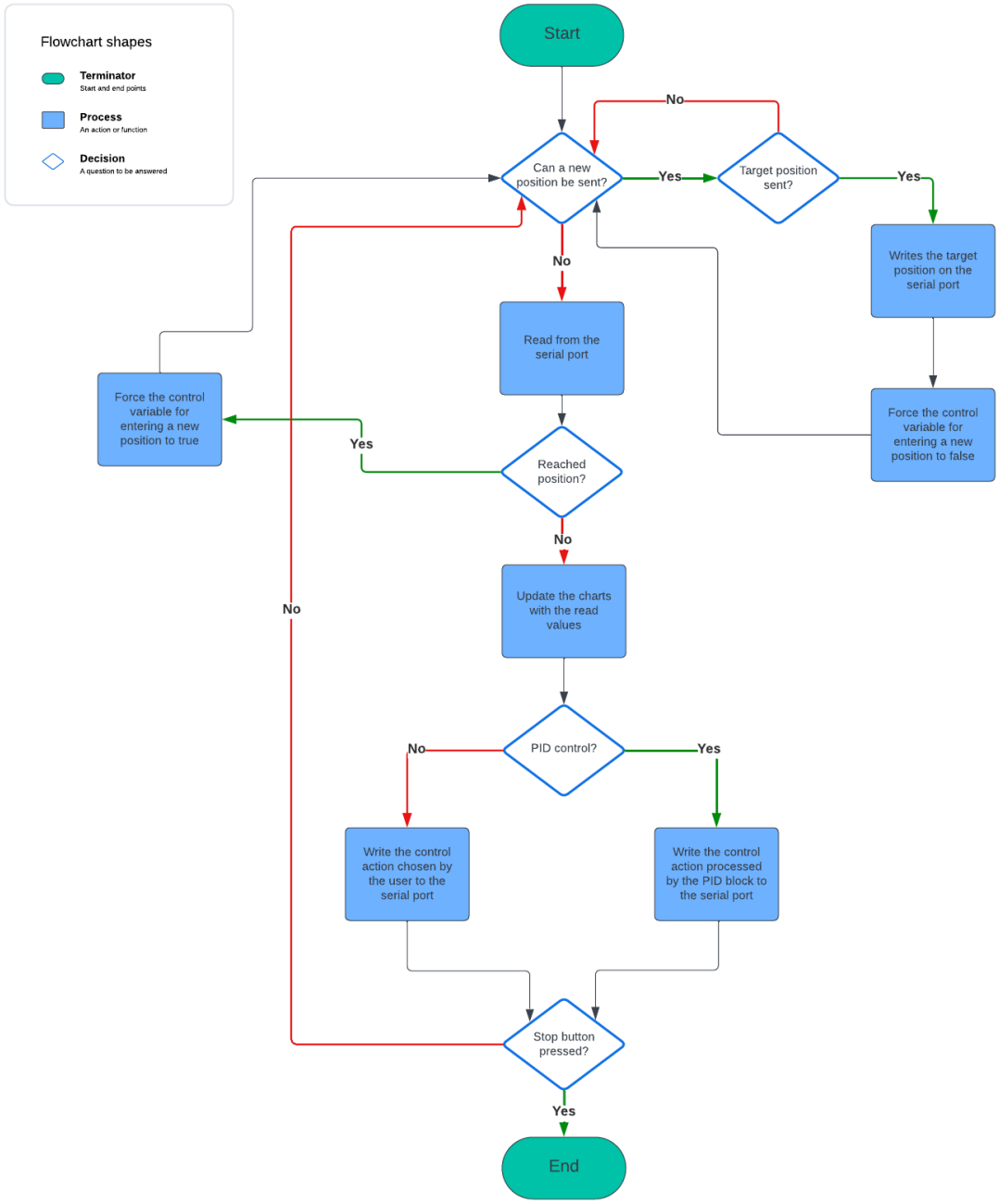

Figure 7: LabVIEW interface flowchart.

3 Results

In this section, we will present the results obtained from the project regarding the position control of a linear actuator using an ultrasonic sensor. Specifically, we will analyze the behavior observed during the application of a user-selected velocity and during the implementation of a velocity determined by the PID controller. Additionally, we will examine how the position graphs vary with changes in the PID parameters. Subsequently, we will assess the performance achieved to determine whether the interface can process data in real-time or if a delay is necessary. Furthermore, we will also demonstrate the standard deviation obtained from multiple measurements taken with the ultrasonic sensor at different distances from the linear actuator. This analysis aims to ascertain the precision of the measurements obtained from the ultrasonic sensor.

3.1 Comparison between manual and automatic speed control

In this subsection, a comparison of the linear actuator's behavior will be conducted between manual speed control and automatic control using a PI controller. Additionally, the influence of variations in the derivative action of the PID controller on the actuator's behavior will be illustrated.

It's advisable to specify that, in this section, graphs related to decreasing position values will be examined. Therefore, only results obtained for a single direction of rotation of the linear actuator motor have been considered. This choice was made to ensure visual consistency within the documentation. However, the same results and considerations apply equally in the case of increasing position values and therefore for the opposite direction of motor rotation.

As shown in Figure 8, the developed LabVIEW interface allows for selecting between two modes of controlling the actuator motor speed. The manual mode enables setting three different speeds, which are activated at specific distance points detected by the ultrasonic sensor. Alternatively, a PI (Proportional-Integral) controller can be used, offering a smoother transition. The PI controller modulates the motor speed to progressively reduce the speed as the actuator approaches the target position.

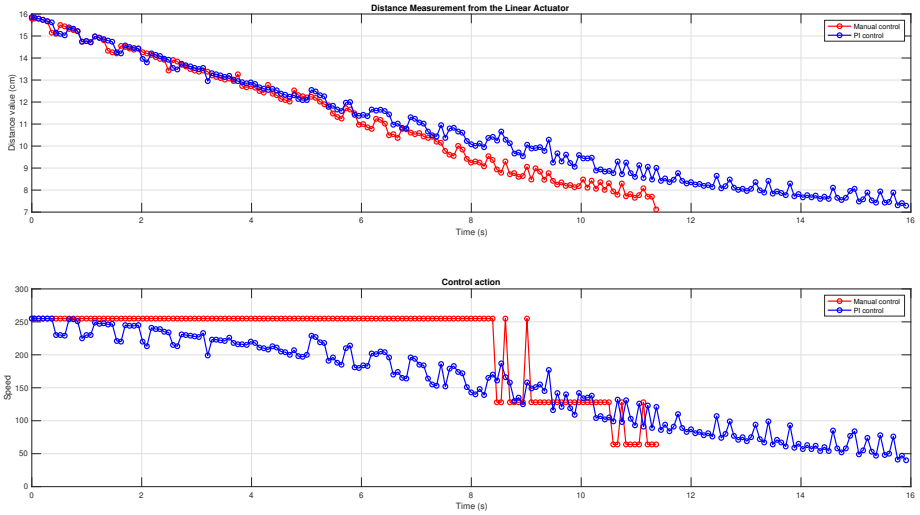

Figure 8: Comparison between distance and velocity values obtained with manual control and with a PI controller, with Kp=60 and Ti = 0.2.

Figure 9 shows the behavior of the position and velocities calculated by the PID controller as the derivative coefficient varies. It can be observed that with a PI controller, which has a zero derivative coefficient, the oscillations are contained, as is also the case with a PID controller with $T_d = 0.001$. However, when higher T_d values are used, such as $T_d = 0.01$ or $T_d = 0.1$, the oscillations become excessive and intolerable.
Increasing the derivative coefficient to high values, such as 0.1, can cause excessive system reactivity initially, leading to pronounced oscillations. This behavior occurs because the derivative term amplifies rapid variations in the error signal. Consequently, the controller reacts disproportionately to initial changes, causing significant oscillations.
When the derivative coefficient is too high, the control signal can saturate, reaching the maximum speed allowed by the system. Although the error decreases as the distance diminishes, the control action remains fixed at the maximum value, representing the motor speed. This happens because the derivative action generates overly intense correction commands, causing the controller to maintain maximum speed in an attempt to quickly correct the perceived error.

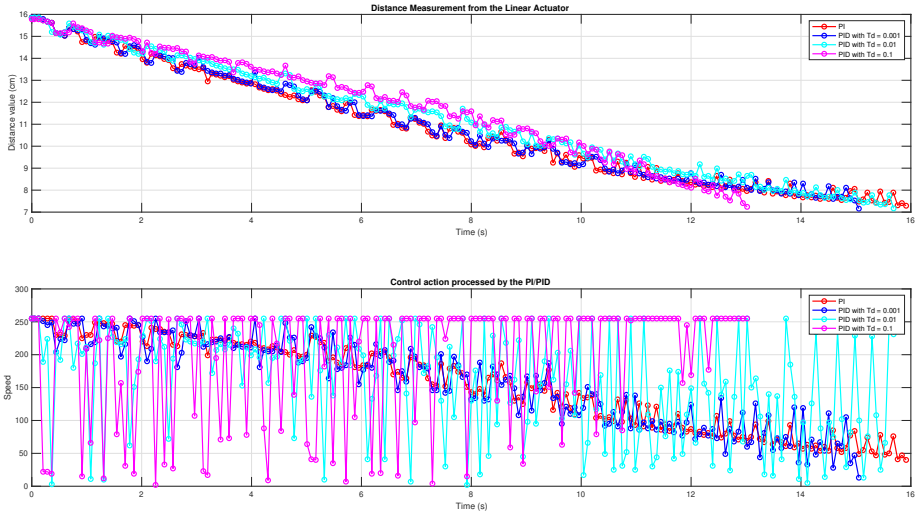

Figure 9: Comparison between distance and velocity values obtained with manual control and with a PI controller, with Kp=60, Ti = 0.2 and variable T_d.

It is further specified that the PI controller used has been empirically tuned following the steps outlined below. We begin by disabling the integral and derivative terms, thus starting with a P controller. The proportional gain is gradually increased until the system responds quickly to reference changes without oscillations. Subsequently, a small integral gain is introduced and gradually increased until the system reaches the setpoint steadily. Finally, if necessary, a small derivative gain is added to reduce oscillations and improve transient response. In the case of this project, it has been observed that optimal performance is achieved even with a PI controller.

3.2 Performance analysis

Within this subsection, an analysis of the performance of the developed interface will be conducted. Specifically, a comparison has been made between the time required by the microcontroller to calculate each distance value sent via the serial port and the reading times of the LabVIEW interface. This comparison was conducted considering variations in the baud rate, which defines the data transmission speed between devices and affects the maximum frequency at which data can be sent

through the serial port.

Three different baud rate values were used: 2400, 9600, and 115200. From the graphical analyses shown in Figures 10, 11, and 12, it is evident that as the baud rate increases, a smaller delay is observed between the time values of the microcontroller and those of the interface.

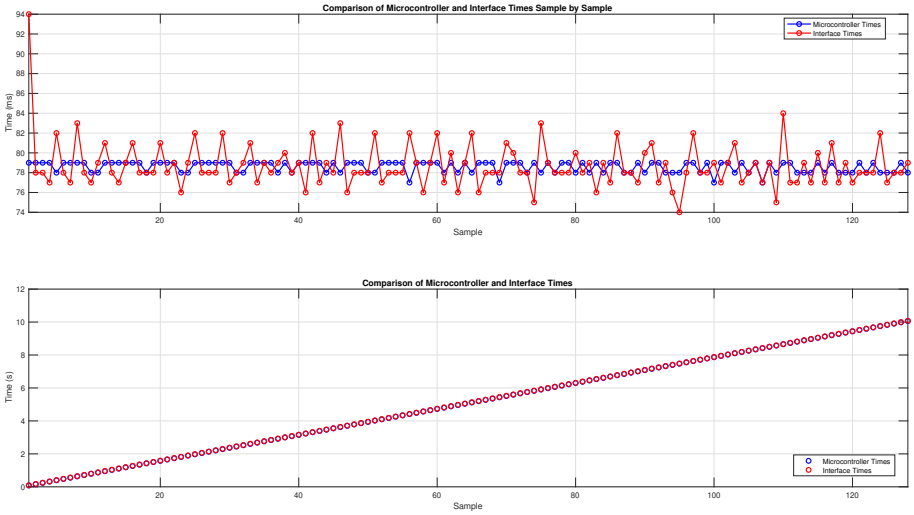

Figure 10: Comparison of MCU and Interface Times with 2400 baud rate and rms_delay = 1.9562.

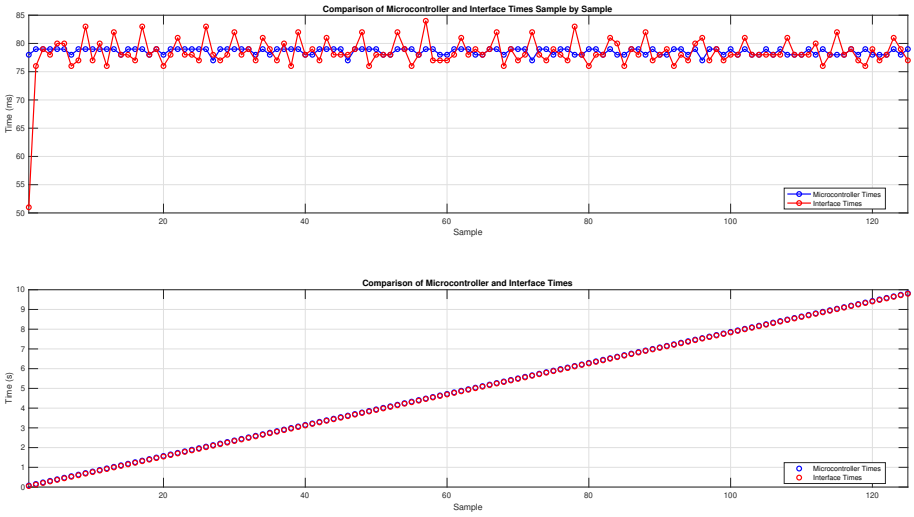

Figure 11: Comparison of MCU and Interface Times with 9600 baud rate and rms_delay = 1.9386.

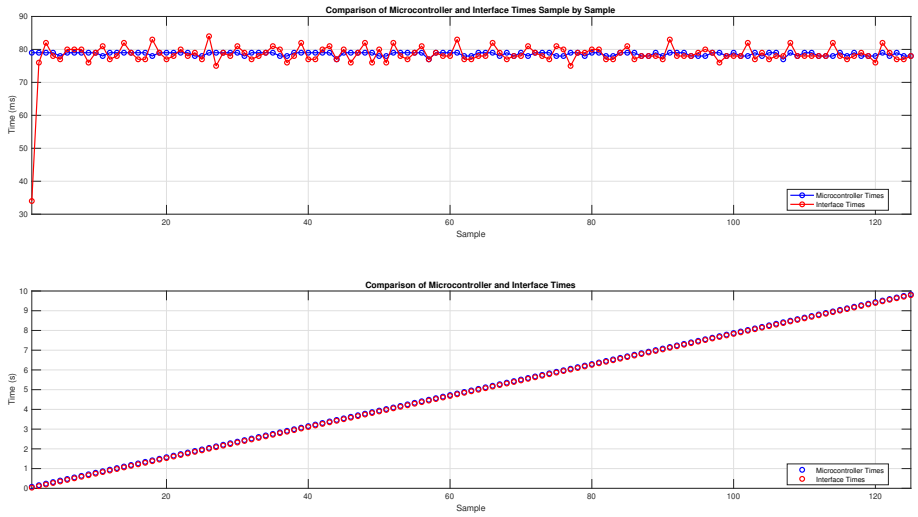

Figure 12: Comparison of MCU and Interface Times with 115200 baud rate and rms_delay = 1.8901.

However, it is observed that for the first values, particularly the first one, in cases of high baud rates such as 9600 and 115200, the interface exhibits a sort of anticipation. It is important to note that this does not imply that the interface receives the first sample after such a time interval, as there is an initial offset between the moment the target position value is transmitted from the LabVIEW interface to the microcontroller and the moment the interface reads the first distance value calculated by the microcontroller. This is due to the fact that the LabVIEW interface's time axis starts from the moment it reads the first distance value from the serial port. Despite this initial offset, it is observed that, excluding the first value which is not representative, the RMS (root mean square) values decrease as the baud rate increases, indicating less dispersion of delays.

For this specific interface, the baud rate value was set to 9600, as it represents a commonly used standard and ensures a satisfactory user experience, thus guaranteeing minimal perception of this inevitable delay.

3.3 Precision analysis

Precision is defined as the difference between an individual measurement and the mean value, expressed by the equation:

$$precision = x_i - \bar{x} \tag{1}$$

This concept stems from the nature of conducting tests on presumed identical materials, under theoretically identical circumstances, where measurements are not always identical. This discrepancy is attributed to the inevitable existence of random errors, known as variance, associated with the measurement method employed. Therefore, this section of the documentation focuses on the precision of the adopted measuring instrument, the HC-SR04.
It is important to introduce two fundamental concepts to address this issue. Firstly, variance is a measure of spread that quantifies the distribution of a random variable. Secondly, standard deviation, defined as the square root of the variance and also known as the root mean square deviation, is calculated using the following formula:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}} \tag{2}$$

Where $\bar{x}$ represents the mean of the measurements and n indicates the total number of measurements taken. Consequently, a smaller standard deviation indicates greater precision, as measurements tend to cluster around the mean. Conversely, a larger standard deviation denotes lower precision of the measuring instrument.

With that said, a table will now be presented (Table 1) containing the target position values and their respective standard deviations, calculated from a sample size of 100 measurements. To accomplish this, a dedicated program implemented on the microcontroller was utilized. This program, upon reaching the user-input target position, conducts 100 measurements at the arrival position and transmits these 100 readings and the standard deviation over the serial port. Subsequently, a Python code retrieves and stores this data in a dataset, facilitating its utilization for the production of graphical representations. These codes are provided and can be found in the appendix of the documentation.

Targ.Pos	7	7.5	8	8.5	9	9.5	10	10.5	11	11.5	12	12.5	13	13.5	14	14.5	15	15.5	16
StD	0.05	0.05	0.05	0.14	0.21	0.17	0.05	0.05	0.05	0.06	0.05	0.05	0.05	0.05	0.06	0.04	0.04	0.11	0.23

Table 1: Table of target position values and their standard deviations

The measure of dispersion can also be interpreted from a boxplot. Indeed, the greater the width between the third and first quartiles in a boxplot, the higher the standard deviation.
As evidenced in Figure 13, it can be observed that for target position values associated with lower standard deviations, such as at the target position of 14.5 cm with a standard deviation of 0.04 cm, the distance between the third and first quartiles is reduced. This is in contrast to situations where the target position, for example, 16 cm, has a standard deviation of 0.23 cm, and where the distance between the third and first quartiles is significantly widened.

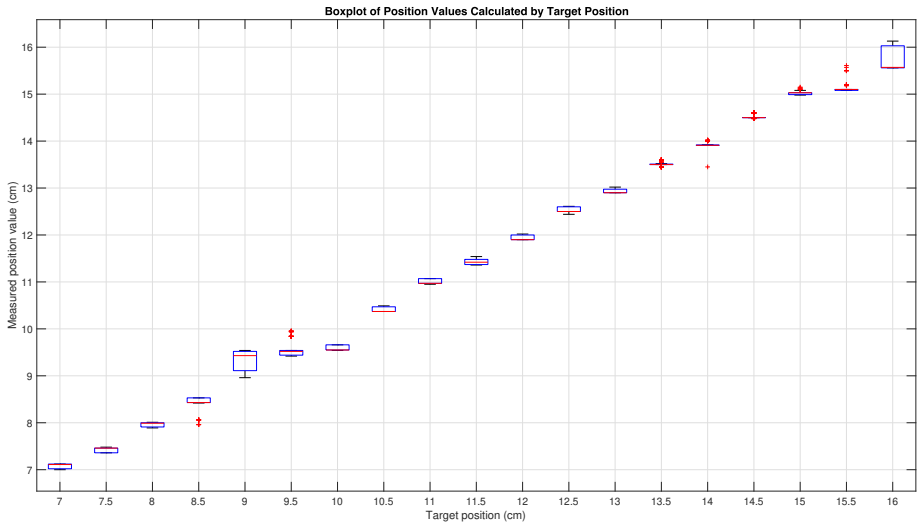

Figure 13: Boxplot of position values calculated by target position.

Despite these slight differences in standard deviation among the various permissible target positions, it is observed that the measuring instrument exhibits, in the worst-case scenario, a standard deviation of 0.23 cm. This justifies the decision to set a higher tolerance level at 0.3 cm for this project. This allows reaching the desired position without oscillations. Empirical testing (Figure 14) has indeed confirmed that with excessively low tolerance values, the actuator tends to oscillate

around the desired position, primarily due to the variance in measurements obtained from the ultrasonic sensor.

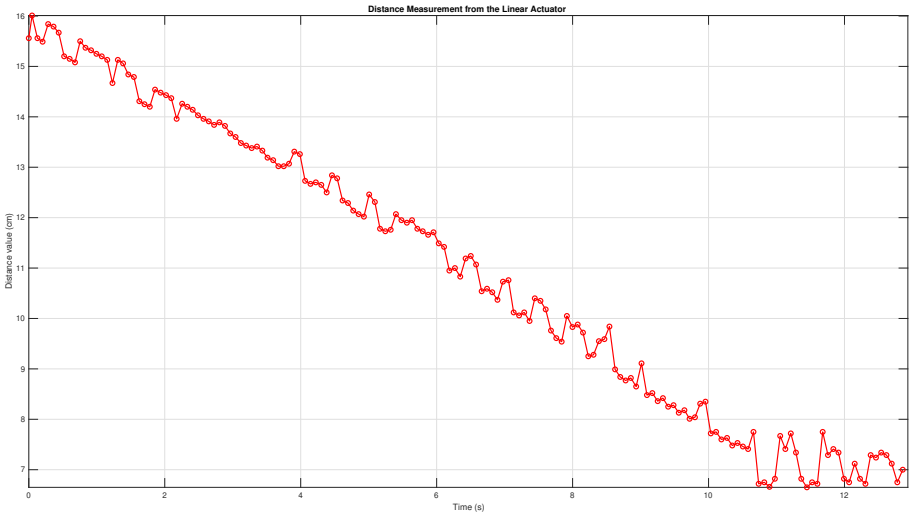

Figure 14: Distance Measurement from the Linear Actuator with max speed and tolerance of 0.1 cm.

A Microcontroller code

Code 1: Main code used for position control of the linear actuator.

```
#include "SR04.h" // Include the SR04 library for the ultrasonic sensor

// Definition of pins
#define RPWM        10
#define LPWM        11
#define ECHO_PIN    12
#define TRIG_PIN    13

// Definition of constant values
const int MAX_SPEED = 255; // Maximum speed for the actuator
const float INITIAL_POSITION = 0.0; // Initial target position
const float TARGET_TOLERANCE = 0.3; // Tolerance for reaching the target position
const float MIN_TARGET_POSITION = 7.0; // Minimum allowed target position
const float MAX_TARGET_POSITION = 16.1; // Maximum allowed target position

SR04 sr04 = SR04(ECHO_PIN, TRIG_PIN); // Initialize the ultrasonic sensor
float distance; // Variable to store the distance measured by the sensor
float targetPosition = INITIAL_POSITION; // Initial target position
float end = 1.0; // Placeholder variable
int Speed = MAX_SPEED; // Current speed of the actuator

unsigned long lastCurrentMillis = 0; // Time of the last distance update
unsigned long currentMillis = 0; // Current time

void setup() {
  Serial.begin(9600); // Initialize serial communication with baud rate 9600
  pinMode(RPWM, OUTPUT); // Set RPWM pin as output
  pinMode(LPWM, OUTPUT); // Set LPWM pin as output
```

```
29  }
30
31  // Function to move the actuator to the desired position
32  void moveActuator(float target) {
33    distance = sr04.Distance(); // Read the current distance from the sensor
34    currentMillis = millis(); // Get the current time
35    Serial.print(currentMillis - lastCurrentMillis); // Print the time elapsed ...
          since the last update
36    lastCurrentMillis = currentMillis; // Update the time of the last update
37    Serial.print(", ");
38    Serial.println(distance); // Print the current distance
39    delay(50); // Delay for stability
40
41    // Loop until the target position is reached within the tolerance
42    while (abs(distance - target) > TARGET_TOLERANCE) {
43      if (distance < target) {
44        // Extend the actuator if the distance is less than the target position
45        analogWrite(RPWM, 0);
46        analogWrite(LPWM, Speed);
47      } else {
48        // Retract the actuator if the distance is greater than the target position
49        analogWrite(RPWM, Speed);
50        analogWrite(LPWM, 0);
51      }
52      // Measure the new distance
53      distance = sr04.Distance();
54      currentMillis = millis(); // Get the current time
55      Serial.print(currentMillis - lastCurrentMillis); // Print the time elapsed ...
            since the last update
56      lastCurrentMillis = currentMillis; // Update the time of the last update
57      Serial.print(", ");
58      Serial.println(distance); // Print the current distance
59
60      // Check if there are new speed commands from the serial input
61      if (Serial.available() > 1) {
62        Speed = Serial.parseInt(); // Parse the new speed value
63      }
64      delay(50); // Delay for stability
65    }
66    Serial.print(end); // Placeholder print
67    Serial.print(", ");
68    Serial.println(end); // Placeholder print
69
70    // Stop the actuator when the target position is reached
71    analogWrite(RPWM, 0);
72    analogWrite(LPWM, 0);
73    // Consume any remaining characters in the serial buffer
74    while (Serial.available() > 0) {
75      Serial.read();
76    }
77  }
78
79  void loop() {
80    if (Serial.available() > 0) {
81      targetPosition = Serial.parseFloat(); // Read the target position from serial ...
            input
82      Speed = MAX_SPEED; // Reset speed to maximum
83      // Consume any remaining characters in the serial buffer
84      while (Serial.available() > 0) {
85        Serial.read();
86      }
87      // Move the actuator only if the target position is within the valid range
88      if (targetPosition ≥ MIN_TARGET_POSITION && targetPosition < ...
            MAX_TARGET_POSITION) {
89        lastCurrentMillis = millis(); // Initialize the time of the last update
```

```
90        moveActuator(targetPosition); // Move the actuator to the target position
91      }
92    }
93  }
```

Code 2: Code used for measuring standard deviations.

```
1   #include "SR04.h"
2   #include <math.h>
3
4   // Definition of pins for motors and ultrasonic sensor
5   #define RPWM        10
6   #define LPWM        11
7   #define ECHO_PIN    12
8   #define TRIG_PIN    13
9
10  // Instantiation of the SR04 object for the ultrasonic sensor
11  SR04 sr04 = SR04(ECHO_PIN, TRIG_PIN);
12
13  // Declaration of global variables
14  float distance; // Variable for distance measured by the sensor
15  float targetPosition = 0.0; // Initial target position
16  float tolerance = 0.05; // Tolerance for reaching the target position
17  int Speed = 128; // Maximum motor speed
18
19  // Setup function for initialization
20  void setup() {
21    Serial.begin(9600); // Initialize serial communication
22    pinMode(RPWM, OUTPUT); // Set RPWM pin as output
23    pinMode(LPWM, OUTPUT); // Set LPWM pin as output
24  }
25
26  // Function to move the actuator to the desired position
27  void moveActuator(float target) {
28    // Measure the current distance from the ultrasonic sensor
29    distance = sr04.Distance();
30    delay(50); // Wait for measurement stability
31
32    // Continue moving the actuator until it reaches the target position within the ...
          specified tolerance
33    while (abs(distance - target) > tolerance) {
34      if (distance < target) {
35        // Extend the actuator if the measured distance is less than the target ...
              position
36        analogWrite(RPWM, 0);
37        analogWrite(LPWM, Speed);
38      } else {
39        // Retract the actuator if the measured distance is greater than the target ...
              position
40        analogWrite(RPWM, Speed);
41        analogWrite(LPWM, 0);
42      }
43      // Measure the distance again after movement
44      distance = sr04.Distance();
45      delay(50); // Wait for measurement stability
46    }
47    // Stop the actuator when the desired position is reached
48    analogWrite(RPWM, 0);
49    analogWrite(LPWM, 0);
50
51    // Calculate the standard deviation based on 100 measurements
52    float measurements[100];
53    float sum = 0;
54    for (int i = 0; i < 100; i++) {
```

```
    measurements[i] = sr04.Distance();
    Serial.println(measurements[i]); // Print measurements for debugging
    sum += measurements[i];
    delay(50); // Wait for measurement stability
  }
  float mean = sum / 100; // Calculate the mean value of measurements
  float variance = 0;
  // Calculate the variance of measurements from the mean
  for (int i = 0; i < 100; i++) {
    variance += pow(measurements[i] - mean, 2);
  }
  // Calculate the standard deviation as the square root of the average variance
  float standardDeviation = sqrt(variance / 100);
  Serial.println(standardDeviation); // Print standard deviation for debugging
}

// Main loop function
void loop() {
  // Check if there are data available on the serial port
  if (Serial.available() > 0) {
    // Read the value of the target position from the serial port
    targetPosition = Serial.parseFloat();
    // Consume any remaining characters in the serial port buffer
    while (Serial.available() > 0) {
      Serial.read();
    }
    // Check if the target position is within the allowed range
    if (targetPosition ≥ 7 && targetPosition < 16.01) {
      // Move the actuator only if the target position is valid
      moveActuator(targetPosition);
    }
  }
}
```

B Python Code

Code 3: Code used to read standard deviations.

```
# This Python script interacts with an Arduino microcontroller through serial communication.
# It sends a target position to the microcontroller, reads 100 distance values and the standard deviation from the microcontroller,
# and then writes these values along with the target position into a CSV file named 'data.csv'.

import serial  # Library for serial communication
import csv     # Library for reading and writing CSV files

# Set the correct serial port and baud rate for communication with the Arduino
ser = serial.Serial('COM3', 9600)  # Change 'COM3' with the correct serial port and 9600 with the baud rate

# Function to send the target position to the Arduino
def send_target_position(target):
    # Converts the target position to a string, appends a newline character, encodes it to bytes, and sends it to the Arduino
    ser.write((str(target) + '\n').encode())

# Function to read 100 distance values from the Arduino
def read_distances():
    # Initializes an empty list to store the distance values
```

```
    distances = []
    # Loops 100 times to read each distance value from the Arduino
    for _ in range(100):
        # Reads a line from the serial port, decodes it from bytes to string, and ... removes any leading/trailing whitespaces
        distance_str = ser.readline().decode().strip()
        # Converts the string distance value to a float and appends it to the ... distances list
        distances.append(float(distance_str))
    # Returns the list of distance values
    return distances

# Function to read the standard deviation from the Arduino
def read_standard_deviation():
    # Reads a line from the serial port, decodes it from bytes to string, and ... removes any leading/trailing whitespaces
    standard_deviation_str = ser.readline().decode().strip()
    # Converts the string standard deviation value to a float and returns it
    return float(standard_deviation_str)

# Main function to execute the script
def main():
    # Loops indefinitely until a valid target position is entered
    while True:
        # Prompts the user to enter a target position within the range 7.0 - 16.0
        target = float(input("Enter the target position (7.0 - 16.0): "))
        # Checks if the entered target position is within the valid range
        if 7.0 ≤ target < 16.01:
            # Sends the target position to the Arduino
            send_target_position(target)
            # Opens 'data.csv' file in append mode for writing
            with open('data.csv', 'a', newline='') as csvfile:
                # Creates a CSV writer object
                csv_writer = csv.writer(csvfile)
                # Reads 100 distance values from the Arduino
                distances = read_distances()
                # Reads the standard deviation from the Arduino
                standard_deviation = read_standard_deviation()
                # Writes the target position, distance values, and standard ... deviation into the CSV file as a row
                csv_writer.writerow([target, distances, standard_deviation])
                # Prints the distance values and standard deviation for the user
                print("Distance values:", distances)
                print("Standard deviation:", standard_deviation)
            # Breaks out of the loop once the data is successfully recorded
            break
        else:
            # Prints a message for an invalid target position and continues the loop
            print("Invalid target position. Please try again.")

# Entry point of the script, calls the main function
if __name__ == "__main__":
    main()
```